Constraint Cascading Style Sheets for the Web

Greg J. Badros
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA
gjb@cs.washington.edu

Alan Borning
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA
borning@cs.washington.edu

Kim Marriott
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria 3168, Australia
marriott@cs.monash.edu.au

Peter Stuckey
Dept. of Computer Science and Software Engineering
University of Melbourne
Parkville, Victoria 3052, Australia
pjs@cs.mu.oz.au

ABSTRACT

Cascading Style Sheets have been introduced by the W3C as a mechanism for controlling the appearance of HTML documents. In this paper, we demonstrate how constraints provide a powerful unifying formalism for declaratively understanding and specifying style sheets for web documents. With constraints we can naturally and declaratively specify complex behaviour such as inheritance of properties and cascading of conflicting style rules. We give a detailed description of a constraint-based style sheet model, CCSS, which is compatible with virtually all of the CSS 2.0 specification. It allows more flexible specification of layout, and also allows the designer to provide multiple layouts that better meet the desires of the user and environmental restrictions. We also describe a prototype extension of the Amaya browser that demonstrates the feasibility of CCSS.

KEYWORDS: Constraints, HTML, style sheets, CSS, Cascading Style Sheets, CCSS, World Wide Web, page layout, Cassowary

INTRODUCTION

Since the inception of the Web there has been tension between the “structuralists” and the “designers.” On one hand, structuralists believe that a Web document should consist only of the content itself and tags indicating the logical structure of the document, with the browser free to determine the document’s appearance. On the other hand, designers (understandably) want to specify the exact appearance of the document rather than leaving it to the browser.

With the recent championing of *style sheets* by the World-Wide-Web Consortium (W3C), this debate has resulted in a compromise. The web document proper should contain the content and structural tags, together with a link to one or more style sheets that determine how the document will be displayed. Thus, there is a clean separation between document structure and appearance, yet the designer has considerable control over the final appearance of the document. W3C has introduced *Cascading Style Sheets*, first CSS 1.0 and now CSS 2.0, for use with HTML documents.

Despite the clear benefits of cascading style sheets, there are several areas in which the CSS 2.0 standard can be improved.

- The designer lacks control over the document’s appearance in environments different from her own. For example, if the document is displayed on a monochrome display, if fonts are not available, or if the browser window is sized differently, then the document’s appearance will often be less than satisfactory.
- CSS 2.0 has seemingly ad hoc restrictions on layout specification. For example, a document element’s appearance can often be specified relative to the parent of the element, but generally not relative to other elements in the document.
- The CSS 2.0 specification is complex and sometimes vague. It relies on procedural descriptions to understand the effect of complex language features, such as table layout. This makes it difficult to understand how features interact.
- Browser support for CSS 2.0 is still limited. We conjecture that this is due in part to the complexity of the specification, but also because the specification does not suggest a unifying implementation mechanism.

We argue that constraint-based layout provides a solution to all of these issues, because constraints can be used to specify *declaratively* the desired layout of a web document. They allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. They also provide a uniform mechanism for understanding layout and cascading. Finally, constraint solving technology provides a unifying implementation technique.

We describe a constraint-based extension to CSS 2.0, called *Constraint Cascading Style Sheets* (CCSS). The extension allows the designer to add arbitrary linear arithmetic constraints to the style sheet to control features such as object placement, and finite-domain constraints to control features such as font properties. Constraints may be given a strength, reflecting their relative importance. They may be used in style rules in which case rewritings of the constraint are created for each applicable element. Multiple style sheets are available for the same media type (e.g., paper vs. screen) with preconditions on the style sheets determining which are appropriate for a particular environment and user requirements.

Our main technical contributions are:

- A demonstration that constraints provide a powerful unifying formalism for declaratively understanding and specifying CSS 2.0.
- A detailed description of a constraint-based style sheet model, CCSS, which is compatible with virtually all of the CSS 2.0 specification. CCSS is a true extension of CSS 2.0. It allows more flexible specification of layout, and also allows the designer to provide multiple layouts that better meet the desires of the user and environmental restrictions.
- A prototype extension of the Amaya browser that demonstrates the feasibility of CCSS. The prototype makes use of the sophisticated constraint solving algorithm Cassowary [4] and a simple one-way binary acyclic finite-domain solver based on BAFSS [12].

BACKGROUND

Cascading Style Sheets (CSS 1.0 in 1997 and CSS 2.0 in 1998) were introduced by the W3C in association with the HTML 4.0 standard. In this section we review relevant aspects of CSS 2.0 [6] and HTML 4.0 [9].

CSS 2.0 and HTML 4.0 provide a comprehensive set of “style” properties for each type of HTML tag. By setting the value of these properties the document author can control how the browser will display each element. Broadly speaking, properties either specify how to position the element relative to other elements, e.g. `text-indent`, `margin`, or `float`, or how to display the element itself, e.g. `font-size` or `color`.

Although the author can directly annotate elements in the document with style properties, CSS encourages the author to place this information in a separate style sheet and then link or import that file. Thus, the same document may be displayed using different style sheets and the same style sheet may be used for multiple documents, easing maintenance of a uniform look for a web site.

A style sheet consists of *rules*. A rule has a *selector* that specifies the document elements to which the rule applies, and *declarations* that specify the stylistic effect of the rule. The declaration is a set of *property*/*value* pairs. Values may be either absolute or relative to the parent element’s value.

```
<HTML> <HEAD>
  <TITLE>Simple Example</TITLE>
  <LINK REL="stylesheet"
        HREF="simple.css"
        TYPE="text/css"> </HEAD>
 <BODY>
  <H1 ID=h>Famous Quotes</H1>
  <P ID=p> At a party at Blenheim Palace,
           Lady Astor said to
           Winston Churchill:
   <BLOCKQUOTE ID=q1>
    If I were married to you, I’d put
    poison in your coffee. And he responded:
    <BLOCKQUOTE ID=q2>
     If you were my wife, I’d drink it.
    </BLOCKQUOTE>
   </BLOCKQUOTE>
  </P>
 </BODY>
</HTML>
```

Figure 1: Example HTML Document

For instance, the style sheet has three rules. The first uses the selector `H1` to indicate that it applies to all elements with tag `H1` and specifies that those first-level headings should be displayed using a 13 pt font. The second rule specifies that paragraph elements should use an 11 pt font. The third rule specifies the appearance of text in a `BLOCKQUOTE`, specifying that the font-size should be 90% of that of the surrounding element.

```
H1 { font-size: 13pt }
P { font-size: 11pt }
BLOCKQUOTE { font-size: 90% }
```

Figure 2: `simple.css`

We can use this style sheet to specify the appearance of the HTML document shown in Figure 1. Notice the link to the style sheet and that we have included an `ID` attribute for all elements since we will refer to them later.[1]

Selectors come in three main flavors: *type*, *attribute*, and *contextual*. These may be combined to give more complex selectors. We have already seen examples of a type selector in which the document elements are selected by giving the “type” of their tag. For example, the type-selector `H1` refers to all first-level heading elements. The wildcard type, “`*`”, matches all tags.

Attribute selectors choose elements based on the values of two attributes that each element in the document tree may

[1] Marking all elements with `ID` attributes defeats the modularity and reuse benefits of CSS; we over-use `ID` tags here strictly as an aid to discussing our examples.

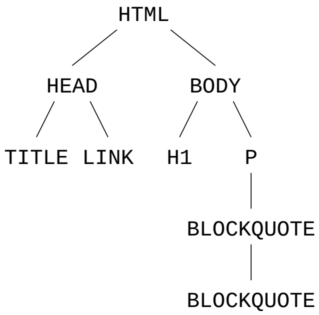


Figure 3: Document tree for the HTML of Figure 1.

optionally provide: CLASS and ID. Multiple elements may specify the same CLASS value, while the ID value should be unique.

Selection based on the CLASS and ID attributes provides considerable power. By using CLASS attributes and selectors, the author can categorize various document elements into groups and then apply different formatting to each of the groups. Similarly, by using ID attributes and selectors, the author can single out document elements for special formatting and then refer to them from the style sheet. Elements with a specific class value are selected using the syntax "*.class*", while instance ids are selected with "*#id*".

Contextual selectors allow the author to take into account where the element occurs in the document, i.e. its context. They are based on the document's *document tree*, which captures the recursive structure of the tagged elements. A context selector allows selection based on the element's ancestors in the document tree.

For instance, the preceding document has the document tree shown in Figure 3. If we want to ensure the innermost block quote does not have its size reduced relative to its parent, we could use

```
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

Less generally, we could individually override the font size for the second BLOCKQUOTE by using a rule with an ID selector:

```
#Q2 { font-size: 100% }
```

Many style properties are inherited by default from the element's parent in the document tree. Generally speaking, properties that control the appearance of the element itself, such as font-size, are inherited, while those that control its positioning are not.

As another example, consider the HTML document shown in Figure 4. We can use a style sheet to control the width of the columns in the table. For example, table.css (Figure 5) contains rules specifying that the elements of the classes medcol and thincol have widths 30% and 20% of their parent tables, respectively. (Note the use of the class selector "." syntax).

```
<HTML> <HEAD>
  <TITLE>Table Example</TITLE>
  <LINK REL="stylesheet"
        HREF="table.css"
        TYPE="text/css"> </HEAD>
 <BODY>
  <TABLE ID=t>
   <COL ID=c1 CLASS=medcol>
   <COL ID=c2>
   <COL ID=c3 CLASS=thincol>
   <TR>
    <TD COLSPAN=2>
     <IMG ID=i1 SRC="back.gif"></TD>
    <TD><IMG ID=i2 SRC="next.gif"></TD>
   </TR>
   <TR>
    <TD>Text1</TD>
    <TD>Text2</TD>
    <TD>Text3</TD>
   </TR>
  </TABLE>
 </BODY>
</HTML>
```

Figure 4: Example HTML Document

```
.medcol { width: 30% }
.thincol { width: 20% }
```

Figure 5: table.css

One of the key features of CSS is that it allows multiple style sheets for the same document. Thus a document might be displayed in the context of the author's special style sheet for that document, a default company style sheet, the user's style sheet and the browser's default style sheet. This is handled in CSS by *cascading* the style sheets, permitting each of the sheets to affect the final rendering.

Cascading, inheritance, and multiple style sheet rules matching the same element may mean that there are conflicts among the rules as to what value a particular style property for that element should take. The exact details of which value is chosen are complex. Within the same style sheet, inheritance is weakest, and rules with more specific selectors are preferred to those with less specific selectors. For instance, each of the rules

```
#Q2 { font-size: 100% }
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

is more specific than

```
BLOCKQUOTE { font-size: 90% }
```

Among style sheets, the values set by the designer are preferred to those of the user and browser, and for otherwise equal conflicting rules, those in a style sheet that is imported or linked first have priority over those subsequently imported or linked. However, a style sheet author may also annotate rules with the strength `!important`, which will override this behavior. In CSS 2.0, for rules designated with strength `!important`, user-specified rules take priority over designer-specified rules.[2]

Despite its power, CSS 2.0 still has a number of limitations. One limitation is that a style property may only be relative to the element's parent, not to other elements in the document. This can result in clumsy specifications, and makes some reasonable layout constraints impossible to express. For example, it is not possible to require that all tables in a document have the same width, and that this should be the smallest width that allows all tables to have reasonable layout. With CSS 2.0, one can only give the tables the same fixed size or the same fixed percentage width of their parent element.

The other main limitation is that it is difficult for the designer to write style sheets that degrade gracefully in the presence of unexpected browser and user limitations and desires. For instance, the author has little control over what happens if the desired fonts sizes are not available. Consider the style sheet `simple.css` again. Imagine that only 10 pt, 12 pt, and 14 pt fonts are available. The browser is free to use 12 pt and 10 pt for headings and paragraphs respectively, or 14 pt and 12 pt, or even 12 pt and 12 pt. Part of the problem is that rules always give definite values to style properties. When different style sheets are combined only one rule can be used to compute the value. Thus a rule is either on or off, leading to discontinuous behavior when style sheets from the author and user are combined. For instance, a sight-impaired user might specify that all font sizes must be greater than 11 pt. However, if the designer has chosen sufficiently large fonts, the user wishes to use the designer's size. This is impossible in CSS 2.0.

CONSTRAINT CASCADING STYLE SHEETS

Our solution to these problems is to use constraints for specifying layout. A constraint is simply a statement of a relation (in the mathematical sense) that we would like to have hold. Constraints have been used for many years in interactive graphical applications for such things as specifying window and page layout. They allow the designer to specify *what* are the desired properties of the system, rather than *how* these properties are to be maintained. The major advantage of using constraints is that they allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. In this section, we describe our constraint-based extension to CSS 2.0, called *Constraint Cascading Style Sheets* (CCSS).

One complication in the use of constraints is that they may conflict. To allow for this we use the *constraint hierarchy* formalism [3]. A constraint hierarchy consists of a collection of constraints, each labeled with a strength. There is a distinguished strength labeled REQUIRED: such constraints must be satisfied. The other strengths denote preferences. There can be an arbitrary number of such strengths, and constraints with stronger strengths are satisfied in preference to ones with weaker strengths. Given a system of constraints, the constraint solver must find a solution to the variables that satisfies the required constraints exactly, and that satisfies the preferred constraints as well as possible, giving priority to those more strongly preferred. The choice of solution depends on the comparator function used to measure how well a constraint is satisfied. In our examples we shall assume the *weighted-sum-better* comparator that sums the errors in satisfying each of the constraints, weighting each error by the strength of that constraint. By using an appropriate set of strength labels we can model the behavior of CSS 2.0.

A Constraint View of CSS 2.0

Hierarchical constraints provide a simple, unifying way of understanding much of the CSS 2.0 specification. This viewpoint also suggests that constraint solvers provide a natural implementation technique. Each style property and the placement of each element in the document can be modeled by a variable. Constraints on these variables arise from browser capabilities, default layout behavior arising from the type of the element, from the document tree structure, and from the application of style rules. The final appearance of the document is determined by finding a solution to these constraints.

The first aspect of CSS 2.0 we consider is the placement of the document elements (i.e., page layout). This can be modeled using linear arithmetic constraints. To illustrate this, we examine table layout—one of the most complex parts of CSS 2.0. The key difficulty in table layout is that it involves information flowing bottom-up (e.g. from elements to columns) and top-down (e.g. from table to columns). The CSS 2.0 specification is procedural in nature, detailing how this occurs. By using constraints, we can declaratively specify what the browser should do, rather than how to do it. Furthermore, the constraint viewpoint allows a modular specification. For example, to understand how a complex nested table should be laid out, we simply collect the constraints for each component, and the solution to these is the answer. With a procedural specification it is much harder to understand such interaction.

Consider the style sheet `table.css` (Figure 5) and the associated HTML document (Figure 4). The associated layout constraints are shown in Figure 6. The notation $\#\mathrm{id}[\mathrm{prop}]$ refers to the value of the property $prop$ for the presentation element corresponding to the document element with ID id.[3] Since we are dealing with a table, the system automatically

[2] This seemingly-inconsistent relative ordering of the `!important` preferences was changed from CSS 1.0 to guarantee that the user has ultimate control over the appearance of a document.

[3] We use associative array-like syntax for referring to properties of elements to avoid the confusion that the alternative "*.selector*" form would

(1)	$\#t[\text{width}] = \#c1[\text{width}] +$ $\#c2[\text{width}] + \#c3[\text{width}]$	REQUIRED
(2)	$\#c1[\text{width}] \geq \text{width(“Text1”)}$	REQUIRED
(3)	$\#c2[\text{width}] \geq \text{width(“Text2”)}$	REQUIRED
(4)	$\#c3[\text{width}] \geq \text{width(“Text3”)}$	REQUIRED
(5)	$\#c3[\text{width}] \geq \#i2[\text{width}]$	REQUIRED
(6)	$\#c1[\text{width}] + \#c2[\text{width}] \geq \#i1[\text{width}]$	REQUIRED
(7)	$\#t[\text{width}] = 0$	WEAK
(8)	$\#c1[\text{width}] = 0.3 * \#t[\text{width}]$	DESIGNER
(9)	$\#c3[\text{width}] = 0.2 * \#t[\text{width}]$	DESIGNER

Figure 6: Example layout constraints

creates a constraint (1) relating the column widths and table width.[4] Similarly, there are automatically created constraints (2-6) that each column is wide enough to hold its content, and (7) that the table has minimal width. Constraints (8) and (9) are generated from the style sheet. Notice the different constraint strengths: from weakest to strongest they are WEAK, DESIGNER and REQUIRED. Since REQUIRED is stronger than DESIGNER, the column will always be big enough to hold its contents. The WEAK constraint `#t[width] = 0` cannot be satisfied exactly; the effect of minimizing its error will be to minimize the width of the table, but not at the expense of any of the other constraints.

These constraints provide a declarative specification of what the browser should do. This approach also suggests an implementation strategy: to lay out the table, we simply use a linear arithmetic constraint solver to find a solution to the constraints. The solver implicitly takes care of the flow of information in both directions, from the fixed widths of the images upward, and from the fixed width of the browser frame downward.

Linear arithmetic constraints are not the only type of constraints implicit in the CSS 2.0 specification. There are also constraints over properties that can take only a finite number of different values, including font size, font type, font weight, and color. Such constraints are called *finite domain* constraints and have been widely studied by the constraint programming community [14]. Typically, they consist of a *domain constraint* for each variable giving the set of values the variable can take (e.g., the set of font sizes available) and required arithmetic constraints over the variables.

As an example, consider the constraints arising from the document in Figure 1 and style sheet `simple.css` (Figure 2). The corresponding constraints are shown in Figure 7. The domain constraints (1-4) reflect the browser's available fonts. The remaining constraints result from the style sheet rules. Note that the third rule generates two constraints (7) and (8), one for each block quote element.

(1)	$\#h[\text{font-size}] \in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(2)	$\#p[\text{font-size}] \in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(3)	$\#q1[\text{font-size}] \in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(4)	$\#q2[\text{font-size}] \in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(5)	$\#h[\text{font-size}] = 13$	DESIGNER
(6)	$\#p[\text{font-size}] = 11$	DESIGNER
(7)	$\#q1[\text{font-size}] = 0.9 * \#p[\text{font-size}]$	DESIGNER
(8)	$\#q2[\text{font-size}] = 0.9 * \#q1[\text{font-size}]$	DESIGNER

Figure 7: Example finite domain constraints

$\#h[\text{font-size}] = 13$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#p[\text{font-size}] = 11$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q1[\text{font-size}] = 0.9 * \#p[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q2[\text{font-size}] = 0.9 * \#q1[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q2[\text{font-size}] = 1.0 * \#q1[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 2\rangle$

Figure 8: Example of overlapping rules

Both of the preceding examples have carefully avoided one of the most complex parts of the CSS 2.0 specification: what to do when multiple rules assign conflicting values to an element's style property. As discussed earlier, there are two main aspects to this: cascading several style sheets, and conflicting rules within the same style sheet.

We can model both aspects by means of hierarchical constraints. To do so we need to refine the constraint strengths we have been using. Apart from REQUIRED, each strength is a lexicographically-ordered tuple

$$\langle cs, i, c, t\rangle.$$

The first component in the tuple, cs, is the *constraint importance* and captures the author-suggested strength of the constraint and its position in the cascade. The constraint importance is one of WEAK, BROWSER, USER, DESIGNER, DESIGNER-IMPORTANT, or USER-IMPORTANT (ordered from weakest to strongest). The importance WEAK is used for automatically generated constraints only. The last three components in the tuple capture the specificity of the rule that generated the constraint: i is the number of `ID` attributes, c is the number of `CLASS` attributes, and t is the number of tag names in the rule (i.e., the depth of the contextual selection).

As an example, consider the constraints arising from the document in Figure 1 with the style sheet

```
H1 { font-size: 13pt }
P { font-size: 11pt }
BLOCKQUOTE { font-size: 90% }
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

The constraints and their strengths for those directly generated from the style sheet rules are shown in Figure 8. Because of its greater weight, the last constraint listed will dominate the second to last one, giving rise to the expected

cause due to CSS's pre-existing use of “.” as a class-name prefix in selectors of rules.

[4] For simplicity, we ignore margins, borders and padding in this example.

$$\begin{array}{ll}
\text{BODY}[\text{font-size}] = 12 & \langle \text{BROWSER}, 0, 0, 0 \rangle \\
\#h[\text{font-size}] = \text{BODY}[\text{font-size}] & \langle \text{WEAK}, 0, 0, 0 \rangle \\
\#p[\text{font-size}] = \text{BODY}[\text{font-size}] & \langle \text{WEAK}, 0, 0, 0 \rangle \\
\#q1[\text{font-size}] = \#p[\text{font-size}] & \langle \text{WEAK}, 0, 0, 0 \rangle \\
\#q2[\text{font-size}] = \#q1[\text{font-size}] & \langle \text{WEAK}, 0, 0, 0 \rangle \\
\#q1[\text{font-size}] = 8 & \langle \text{DESIGNER}, 0, 0, 1 \rangle \\
\#q2[\text{font-size}] = 8 & \langle \text{DESIGNER}, 0, 0, 1 \rangle
\end{array}$$

Figure 9: Example of inheritance rules

behavior—that the longer contextual selection of a blockquote within a blockquote will govern the appearance of those nested blockquotes.

The remaining issue we must deal with is inheritance of style properties such as font size, and the expression of this inheritance within our constraint formalism. For each inherited property, we need to automatically create an appropriate constraint between each element and its parent. At first glance, these should simply be WEAK equality constraints. Unfortunately, this does not model the inherent directionality of inheritance.

For instance, imagine displaying the document in Figure 1 with the style sheet

```
BLOCKQUOTE { font-size: 8pt }
```

where the default font size is 12 pt. The scheme outlined above gives rise to the constraints shown in Figure 9. One possible weighted-sum-better solution to these constraints is that the heading is in 12 pt and the rest of the document (including the paragraph) is in 8 pt. The problem is that the paragraph element $\#p$ has "inherited" its value from its *child*, the BLOCKQUOTE element $\#q1$.

To capture the directionality of inheritance we use *read-only* annotations [3] on variables that represent inherited attributes. Intuitively, a read-only variable v in a constraint c means that c should not be considered until the constraints involving v as an ordinary variable (i.e., not read-only) have been used to compute v's value.

To model inheritance, we need to add the inheritance equalities with constraint importance of WEAK, and mark the variable corresponding to the parent's property as read-only. The read-only annotation ensures that the constraints are solved in an order corresponding to a top-down traversal of the document tree. Thus, the above example modifies the constraints in Figure 9 so that each font size variable on the right hand side has a read-only annotation.

Extending CSS 2.0

We have seen how we can use hierarchical constraints to provide a declarative specification for CSS 2.0. There is, however, another advantage in viewing CSS 2.0 in this light. The constraint viewpoint suggests a number of natural extensions that overcome the expressiveness limitations of CSS 2.0 discussed previously. We call this extension CCSS—Constraint Cascading Style Sheets.

As the above examples indicate, virtually all author and user constraints generated from CSS 2.0 either constrain a style property to take a fixed value, or relate it to the parent's style property value. One natural generalization is to allow more general constraints such as inequalities. Another natural generalization is to allow the constraint to refer to other variables—both variables corresponding to non-parent elements and to "global" variables.

In CCSS, we allow constraints in the declaration of a style sheet rule. The CSS-style `attribute:value` pair is reinterpreted in this context as the constraint `attribute = value`. We prepend all constraint rules with the `constraint` pseudo-property so that CCSS is backwards compatible with browsers supporting only CSS. In a style sheet rule, the constraint can refer to attributes of `parent` and `left-sibling`. For example:

```
P { constraint:
     font-size <=
       (parent[parent])[font-size] + 2 }
```

is a rule that applies constraints that relate the font-size of a paragraph element to the font-size of its grandparent element.

CCSS style sheets also allow the author to introduce global constrainable variables using a new `@variable` directive. A variable identifier is lexically the same as a CSS `ID` attribute. The author can express constraints among global constrainable variables and element style properties using a new `@constraint` directive. There are also various global built-in objects (e.g., `Browser`) with their own attributes that can be used.

These extensions add considerable expressive power. For instance it is now simple to specify that all tables in the document have the same width, and that this is the smallest width that allows all tables to have a reasonable layout:

```
@variable table-width;
TABLE { constraint: width = table-width }
```

Similarly we can specify two columns `c1` and `c2` in the same (or different) tables have the same width (the smallest for reasonably laying out both):

```
@constraint #c1[width] = #c2[width];
```

It also allows the designer to express preferences in case the desired font is not available. For example adding

```
H1 { constraint: font-size >= 13pt }
P  { constraint: font-size >= 11pt }
```

to `simple.css` (Figure 2) will ensure that larger fonts are used if 13 pt and 11 pt fonts are not available.

Finally, a sight-impaired user can express the strong desire to have all font sizes greater than 12 pt:

```
* {constraint: font-size >= 12pt !important}
```

As long as the font size of an element is 12 pt or larger it will not be changed, but smaller fonts will be enlarged.

Note that the style sheet author is not allowed to explicitly specify a constraint to be REQUIRED as this would admit the possibility of an unsatisfiable constraint system. Instead, REQUIRED constraints are generated implicitly for capturing relationships inherent in the structure of the layout, such as a table's width being the sum of the widths of its columns.

Providing inequality constraints allows the author to control the document appearance more precisely in the context of browser capabilities and user preferences. Additionally, CCSS allows the author to give alternate style sheets for the same target media. Each style sheet can list preconditions for their applicability using a new `@precondition` directive. For efficiency, the precondition can only refer to various pre-defined variables. The values of these variables will be known (i.e. they will have specific values) at the time the precondition is tested. For example:

```
@precondition Browser[frame-width] >= 800px;
@precondition ColorMonitor = True;
```

We extend the style sheet `@import` directive to permit listing multiple style sheets per line, and the first applicable sheet is used (the others are ignored). If no style sheet's preconditions hold, none are imported. Consider the example directive

```
@import "wide.css", "tall.css", "small.css";
```

If `wide.css`'s preconditions fail, but `tall.css`'s succeed, the layout uses `tall.css`. If, through the course of the user resizing the top-level browser frame, `wide.css`'s preconditions later become satisfied, the layout does not switch to that style sheet unless `tall.css`'s preconditions are no longer satisfied. That is, the choice among style sheets listed with one directive is only revisited when a currently-used style sheet is no longer applicable.

As an example, consider a style sheet for text with pictures. If the page is wide, the images should appear to the right of the text; if it is narrow, they should appear without text to the left; and if it is too small, the images should not appear at all. This can be encoded as:

```
/* wide.css */
@precondition Browser[frame-width] > 550px;
IMG { float: right}

/* tall.css */
@precondition Browser[frame-width] <= 600px;
@precondition Browser[frame-height] > 550px;
IMG { clear: both; float: none}

/* small.css */
IMG { display: none }
```

Preconditions become even more expressive in the presence of support for CSS positioning [10] and a generalized `flow` property [7].

IMPLEMENTATION

Prototype Web Browser

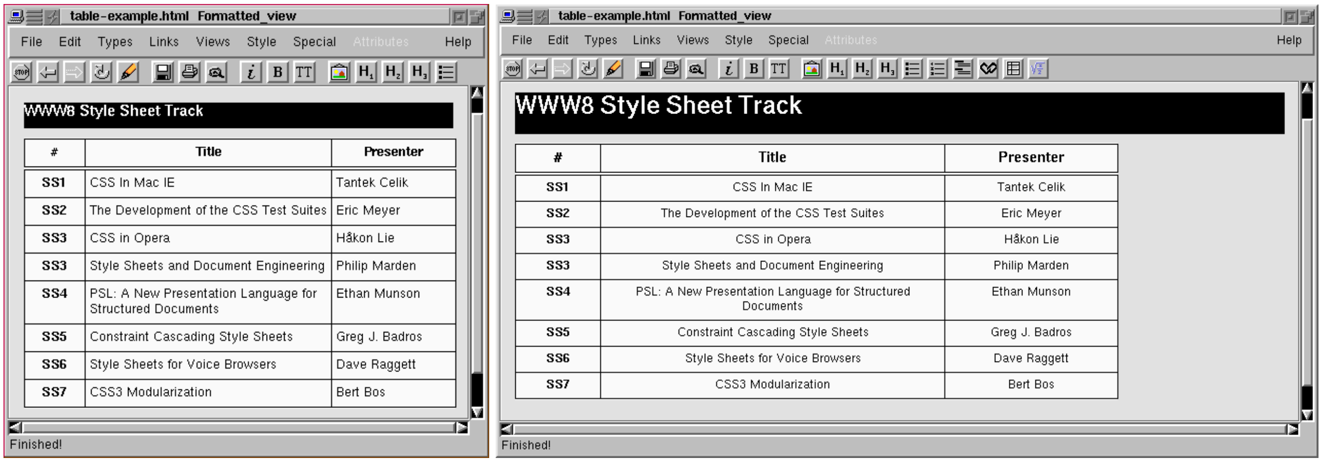
We have implemented a representative subset of CCSS to demonstrate the additional expressiveness it provides to web designers. Our prototype is based on version 1.4a of Amaya [8], the W3 Consortium's browser. Amaya is built on top of Thot, a structured document editor, and has partial support for CSS1. Amaya is exceptionally easy to extend in some ways (e.g., adding new HTML tags), and provides a stable base from which to build.

Our support for constraints in Amaya covers the two main domains for constraints that we have discussed: table widths (for illustrating page layout relationships) and font sizes (for illustrating the solution of systems involving inherited attributes). In our prototype, HTML and CSS statements can contain constraints and declare constrainable variables. In HTML statements, constrainable variables, in addition to specific values, can be attached by name to element attributes (e.g., to the "width" of a table column). When the constraints of the document force the values assigned to variables to change, the browser updates its rendering of the current page, much as it does when the browser window is resized (which often caused the re-solve in the first place).

We have also extended Amaya to support preconditions on style sheets and the generalized "`@import`" CCSS rule. The performance when switching among style sheets is similar to a reload, and when the style sheets are cached on disk, performance is good even when switching style sheets during an interactive resize. (It may be useful to provide background pre-fetching of alternate style-sheets to avoid latency when they are first needed.) See Figure 10 for screen shots of an example using our prototype's support for both table layout and preconditions. As the support for CSS improves in browsers, more significant variations will be possible through the use of our `@precondition` and extended `@import` directives.

We compared the performance of our prototype browser to an unmodified version of Amaya 1.4a, both fully optimized, running on a PII/400 displaying across a 10Mbit network to a Tektronix X11 server on the same subnet. Our test case was a small example on local disk using seven style sheets. We executed 100 re-loads, and measured the total wall time consumed. The unmodified browser performed each re-load and re-render in 190 ms, while our prototype took only 250 ms even when sized to select the last alternative style sheet in each of three `@import` directives. This performance penalty is reasonable given the added expressiveness and features the prototype provides.

One of the most important benefits of re-framing CSS as con-



WWW8 Style Sheet Track

#	Title	Presenter
SS1	CSS In Mac IE	Tantek Celik
SS2	The Development of the CSS Test Suites	Eric Meyer
SS3	CSS in Opera	Håkon Lie
SS3	Style Sheets and Document Engineering	Philip Marden
SS4	PSL: A New Presentation Language for Structured Documents	Ethan Munson
SS5	Constraint Cascading Style Sheets	Greg J. Badros
SS6	Style Sheets for Voice Browsers	Dave Raggett
SS7	CSS3 Modularization	Bert Bos

WWW8 Style Sheet Track

#	Title	Presenter
SS1	CSS In Mac IE	Tantek Celik
SS2	The Development of the CSS Test Suites	Eric Meyer
SS3	CSS in Opera	Håkon Lie
SS3	Style Sheets and Document Engineering	Philip Marden
SS4	PSL: A New Presentation Language for Structured Documents	Ethan Munson
SS5	Constraint Cascading Style Sheets	Greg J. Badros
SS6	Style Sheets for Voice Browsers	Dave Raggett
SS7	CSS3 Modularization	Bert Bos

Figure 10: Screen shots of our prototype browser. In the view on the left, a narrow style sheet is in effect because the browser width ≤ 800 pixels, while on the right a wide style sheet is used. Interactively changing the browser width dynamically switches between these two presentations. In both figures, the first column is $\frac{1}{4}$ the width of the second column, which is twice the width of the last column. On the left, the table consumes 100% of the frame width, but on the right, the table width is the browser width minus 200 pixels. Also notice the changes in font size and text alignment.

straints is that it provides an implementation approach for even the standard CSS features. To simplify our prototype and ensure it remains a superset of CSS functionality, we currently do not treat old-style declarations as constraints, but instead rely on the existing implementation's handling of those rules. However, if designed into a browser from the beginning, treating all CSS rules as syntactic sugar for underlying constraints will result in large savings in code and complexity. The cascading rules would be completely replaced by the constraint solver's more principled assignment of values to variables, and the display engine need only use those provided values, and redraw when the solver changes the current solution.

Constraint Solving Algorithms

While the semantics of the constraints is independent of the algorithms used to satisfy them, for interactive applications such as a web browser, we nevertheless must select an algorithm that is capable of efficiently finding solutions to the constraints at interactive speeds. Our implementation uses two algorithms: Cassowary [4] and a restricted version of BAFSS [12].

The Cassowary algorithm handles linear arithmetic equality and inequality constraints. The collection of constraints may include cycles (i.e. simultaneous equalities and inequalities or redundant constraints) and conflicting preferences. Cassowary is an incremental version of the simplex algorithm, a well-known and heavily studied technique for finding a solution to a collection of linear equality and inequality constraints that minimizes the value of a linear expression called the *objective function*. However, commonly available implementations of the simplex algorithm are not suitable for interactive applications such as a web browser.

Cassowary supports the weighted-sum-better comparator [3] for choosing a single solution from among those that satisfy all the required constraints. As mentioned earlier, this comparator computes the error for a solution by summing the product of the strength tuple and the error for each constraint that is unsatisfied. To model the CSS importance rules in a hierarchy of constraint strengths, we encode the symbolic levels of importance as tuples as well; for example, USER-IMPORTANT is $\langle 1,0,0,0,0,0\rangle$ and BROWSER is $\langle 0,0,0,0,1,0\rangle$. Thus, no matter what scalar error a BROWSER constraint has, it will never be satisfied if doing so would force a USER-IMPORTANT constraint to not be satisfied. Similarly the last three components of the strength tuple are encoded as $\langle 10^i, 10^c, 10^t\rangle$.[5] Since the Cassowary toolkit operates on constraints with strengths that are a single n-tuple, we internally use 9-tuples to represent strengths—for example,

$$\langle 1,0,0,0,0,0,10^0,10^1,10^0\rangle$$

is the strength of a user-specified `!important` constraint whose selector only contains a single class name.

BAFSS uses a dynamic programming approach to handle systems of font constraints which are binary (i.e., a constraint with two variables) and for which the associated constraint graph is acyclic. For the font constraints implied by CSS, we are able to simplify the algorithm because all of the constraints relate a read-only size attribute in the parent element to the size attribute of a child element. Given this additional restriction that all constraints are one-way, the algorithm is

[5] This does not exactly match the CSS specificity rules. For example if the error in a constraint with strength $\langle \text{WEAK},0,0,1\rangle$ is 10 times greater than the error in a conflicting constraint with strength $\langle \text{WEAK},0,0,2\rangle$, the first constraint will affect the final solution. By choosing appropriate error functions we can make this unlikely to occur in practice. However, the more general constraint hierarchy support may actually permit more desirable interactions rather than the strict strength ordering imposed by CSS.

simple: visit the variable nodes in topological order and assign each a value that greedily minimizes the error contribution from that variable.

Both constraint solvers are implemented within the Cassowary Constraint Solving library [1].

RELATED WORK

The most closely related research is our earlier work on the use of constraints for web page layout [5]. This system allowed the web page author to construct a document composed of graphic objects and text. The layout of these objects and the text font size were described in a separate "layout sheet" using linear arithmetic constraints and finite domain constraints. Like CCSS, layout sheets had preconditions, controlling their applicability.

The work reported here, which focuses on how to combine constraint-based layout with CSS, is complementary to our previous research. One of the major technical contributions here is to provide a declarative semantics for CSS based on hierarchical constraints; this issue was not addressed in our prior work [5]. There are two fundamental differences between layout sheets and CCSS. Layout sheets are not style sheets in the sense of CSS since they can only be used with a single document. Constraints only apply to named elements, and there is no concept of a style rule that applies to multiple elements—the constraints that are used are exactly the constraints that the author has specified. The other fundamental difference between the earlier work [5] and CCSS is that the former has no analogue of the document tree. In essence, the document is modeled as a flat collection of objects; there is no notion of inheritance, and nearly all layout must be explicitly detailed in the layout sheet.

Cascading Style Sheets are not the only kind of style sheet. The Document Style Semantics and Specification Language (DSSSL) is an ISO standard for specifying the format of SGML documents. DSSSL is based on Scheme, and provides both a transformation language and a style language. It is very powerful but complex to use. More recently, W3C has begun designing the XSL style sheet for use with XML documents. XSL is similar in spirit to DSSSL. PSL [13] is another style sheet language; its expressiveness lies midway between that of CSS and XSL. The underlying application model for all three is the same: take the document tree of the original document and apply transformation rules from the style sheet in order to obtain the presentation view of the document, which is then displayable by the viewing device. In the case of XSL, the usual presentation view is an HTML document whose elements are annotated with style properties.

None of these other style sheet languages allow true constraints. Extending any of them to incorporate constraints would offer many of the same benefits as it does for CSS, namely, the ability to flexibly combine user, browser, and designer desires and requirements, and a simple powerful model for layout of complex objects, such as tables. The simplest extension is to allow constraints in the presentation view of the document. (Providing constraints in the transformation rules would seem to offer little advantage.) In the case of DSSSL a natural way to do this is to embed a constraint solver into Scheme (as in SCWM [2]). In the case of XSL, since HTML is often used as the targeted visual rendering language, the simplest change is to augment that language to be HTML with CCSS style properties. Then the XSL translator would simply generate HTML and a CCSS style sheet, with a CCSS-enhanced browser still performing the dynamic constraint solving, rendering, and interaction.

Regarding other user interface applications of constraints, there is a long history of using constraints in interfaces and interactive systems, beginning with Ivan Sutherland's pioneering Sketchpad system [17]. Constraints have also been used in several other layout applications. IDEAL [18] is an early system specifically designed for page layout applications. Harada, Witkin, and Baraff [11] describe the use of physically-based modeling for a variety of interactive modeling tasks, including page layout. There are numerous systems that use constraints for widget layout [15, 16], while Badros [2] uses constraints for window layout.

CONCLUSIONS AND FUTURE WORK

We have demonstrated that hierarchical constraints provide a unifying, declarative semantics for CSS 2.0 and also suggest a simplifying implementation strategy. Furthermore, viewing CSS from the constraint perspective suggests several natural extensions. We call the resulting extension CCSS—Constraint Cascading Style Sheets. By allowing true constraints and style sheet preconditions, CCSS increases the expressiveness of CSS 2.0 and, importantly, allows the designer to write style sheets that combine more flexibly and predictably with user preferences and browser restrictions. We have demonstrated the feasibility of CCSS by modifying the Amaya browser. However, substantial work remains to develop an industrial-strength browser supporting full CCSS, in part because of Amaya's lack of support for CSS 2.0. A more complete implementation will be especially useful for investigating the important issue of how well the constraint systems and solver scale to larger, more complicated designs that further exploit our constraint extensions.

Apart from improving the current implementation, we have two principal directions for further extensions to CCSS. The first is to increase the generality and solving capabilities of the underlying solver. For example, style sheet authors should be able to arbitrarily annotate variables as read-only so that they have greater control over the interactions of global variables. Additionally, virtually all CSS properties, such as color and font weight, could be exposed to the constraint solver once we integrate other algorithms into our solving toolkit.

The second extension is to allow “predicate” selectors in style sheet rules. These selectors would permit an arbitrary predicate to be tested in determining the applicability of a rule to an element in the document structure tree. Predicate selectors can be viewed as a generalization of the existing selectors; an `H1 P` selector is applied only to nodes n for which the predicate “n[type] = `P` and $\exists m$ parent-of n such that m[type] = `H1`” holds. These predicate selectors would allow the designer to take into account the attributes of the selected element’s parents and children, thus, for instance, allowing the number of items in a list to affect the appearance of the list (as in an example used to motivate PSL [13]).

A final important area for future work is the design, implementation, and user testing of graphical interfaces for writing and debugging constraint cascading style sheets and web pages that use them.

ACKNOWLEDGMENTS

We thank Bert Bos and Håkon Lie of the CSS group of the W3 Consortium for early feedback on these ideas and our proposed extensions to CSS. This research has been funded in part by a US National Science Foundation Graduate Fellowship for Greg Badros and by NSF Grant No. IIS-9975990, and in part by a grant from the Australian Research Council.

REFERENCES

1. G. Badros and A. Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998.

2. G. Badros and M. Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1997–1999. http://scwm.mit.edu/.

3. A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

4. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.

5. A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, pages 173–182, 1997.

6. B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2. W3C Working Draft, January 1998. http://www.w3.org/TR/WD-css2/.

7. B. Bos, D. Raggett, and H. Lie. Frame-based layout via style sheets. W3C Working Draft. http://www.w3.org/TR/WD-layout.

8. W3 Consortium. Amaya web browser software. Web page, October 1998. http://www.w3.org/Amaya.

9. W3 Consortium. HTML 4.0 specification. Technical report, W3 Consortium, 1998. http://www.w3.org/TR/REC-html40.

10. S. Furman and S. Isaacs. Positioning HTML elements with cascading style sheets. W3C Working Draft. http://www.w3.org/TR/WD-positioning.

11. M. Harada, A. Witkin, and D. Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH ’95 Conference Proceedings*, pages 199–208, Los Angeles, August 1995. ACM.

12. R. Lin, K. Marriott, and P. Stuckey. Flexible font-size specification in Web documents. In *Proceedings of the 22 Australasian Computer Science Conference*, Auckland, New Zealand, January 1999. Springer-Verlag.

13. P. Marden, Jr. and E. Munson. PSL: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, 4(10), 1998. http://www.cs.uwm.edu/~multimedia.

14. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

15. B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, November 1990.

16. B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

17. I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963.

18. C. van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.